Chameleon

Decoupling Instrumentation from Development Tools with Explicit Meta-Events

Masterarbeit
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Andrea Quadri

12. Januar 2012

Leiter der Arbeit:
Prof. Dr. Oscar Nierstrasz
Jorge Ressia
Institut für Informatik und angewandte Mathematik

Abstract

Software instrumentation monitors the run-time behavior of a system to support a particular kind of analysis. The behavior of a system can be understood as a set of meta-level events that occur to achieve a goal. Instrumentation is often realized with the help of reflective mechanisms that reify these meta-events. Despite many advances, these approaches have a common drawback: the instrumentation of the base level is tightly coupled to the behavior of the development tools that required the instrumentation in the first place. This prevents development tools from leveraging the event abstractions used by other tools and forces them to respecify the events and their instrumentation. We propose to resolve this problem by modeling meta-events explicitly. Instrumentation is dedicated to generating meta-events, and is fully separated from analysis tools which selectively subscribe to these events by applying the observer pattern at the meta-level. We survey approaches to reflection to establish the key requirements for practical applications, and illustrate the limitations of these approaches. We introduce *Chameleon,* a prototype tool modeling the meta-level as explicit meta-events observable by development tools.

Acknowledgments

I would like to thank Prof. Oscar Nierstrasz for his guidelines, helpful discussions and inputs. Furthermore I would like to thank my assistant Jorge Ressia for his patience, feedback and support throughout this journey. Last but not least I thank my family and friends for their support.

Contents

1 Introduction **5**

2 Related Work **7**
2.1 Applications of Instrumentation . . . 7
2.2 Structural Reflection . . . 8
2.3 Behavioral Reflection . . . 8
2.3.1 CodA . . . 8
2.3.2 Iguana . . . 9
2.4 Partial Behavioral Reflection . . . 11
2.4.1 Reflex . . . 11
2.4.2 Reflectivity . . . 12
2.5 Aspect-oriented Programming . . . 12
2.6 Aspect-oriented- with Component-based Programming . . . 13
2.7 Event-based Aspect-oriented Programming . . . 14

3 Iguana/S **16**
3.1 Implementation . . . 16
3.2 Findings . . . 18

4 Chameleon **19**
4.1 Events . . . 19
4.2 Instrumentation for Signaling Events . . . 20
4.3 Announcer . . . 21
4.4 Observers . . . 22

5 Validation **23**
5.1 Domain-Polluted Instrumentation . . . 23
5.2 Language-biased Events . . . 25
5.3 Static Instrumentation Scoping . . . 25

6 Implementation **28**
6.1 Managing AST Meta-Objects . . . 28
6.2 Instrumentation Details . . . 29
6.3 Extending Events . . . 30
6.4 Benchmark . . . 31

7 Conclusion **32**
7.1 Separation of Concerns . . . 32
7.2 Dynamic Observer-oriented Scoping . . . 33
7.3 Unbiased Events . . . 33

7.4 Future Work 33

Appendices **39**
A.1 Installation Guide 39

1 Introduction

Instrumentation is the process of adapting a software system to measure run-time attributes of interest. Development tools like profilers, loggers and code coverage analysis tools are traditional applications of instrumentation. Existing approaches to instrumentation are perfectly capable of implementing any of these use cases. Combining analyses, however, poses a number of difficulties.

Domain-Polluted Instrumentation. Existing event-based instrumentation approaches tend to couple the instrumentation behavior with the domain behavior. For example, a message counter profiler needs to instrument an application to reify message sends. If we would like to apply a message send logger to the same application we cannot reuse the already instrumented message send reification since it is coupled to the profiling domain. Due to this, a new instrumentation for the same reification is needed. Existing approaches pollute the instrumentation with domain behavior thus rendering this instrumentation only usable for a particular consumer, domain or context. Instrumentation behavior and its consumers are tightly coupled.

Language-biased Events. Event-based reflective approaches are coupled with specific characteristics of the host language. The language's internals define the number of canonical events and which abstractions they should represent. For example, some aspect-oriented programming (AOP) [Kic96, KLM+97, KIL+97] languages depend on the structure and organization of the code. The programmer must express concerns in terms of code-related events, usually method calls, rather than in terms of domain concepts. This problem is known as the *fragile pointcut problem* [SK04].

Static Instrumentation Scoping. Most instrumentation approaches use statically defined conditions to control the runtime impact of the instrumentation. Which portions of the system should trigger an event is defined through conditions which have to be manually maintained. There is no dynamic mechanism for plugging and unplugging instrumentations on specific objects.

Our approach. We propose to resolve these problems by fully separating instrumentation from analysis with the help of explicit meta-level events. We simplify the meta-level's behavioral model by offering a single canonical event which models the execution of an abstract syntax tree (AST) node. Any other object-related event can be expressed in terms of this canonical event. Objects in an application are

instrumented to reify meta-level events. Analysis tools select which events to observe for the purpose of profiling, logging, coverage, *etc.*

The contributions this paper are:

- A simplified approach to behavioral reflection through operational decomposition. Our approach proposes a single event on top of which any other meta-level event reification can be defined.
- Chameleon, a prototype of our meta-event reflection model.
- An explicit event approach on top of AST manipulation.
- An extensible model for event-based instrumentation.
- A stricter separation of concerns between instrumentation and the consumers of events.
- An instrumentation scoping system based on the consumer's point of view.

Our approach provides enhanced separation of concern capabilities by using runtime objects as the modularity unit. Instrumentation requirements are applied to specific objects thus allowing one to reflect on any portion of the runtime system modeled with objects.

Outline. In chapter 2 we discuss the related work with more details about their different implementations and problems. Then in chapter 3 we show our first prototype and what ideas we gained from it. Chapter 4 shows Chameleon approach in a nutshell. In chapter 5 we show how Chameleon solves the drawback of other approaches. Chapter 6 presents how Chameleon is implemented. In chapter 7 we summarize the paper.

2
Related Work

In this chapter we review the state of the art in instrumentation techniques, while highlighting problems and open issues. In particular, we shall see that existing approaches offer limited expressiveness in terms of the way that run-time events are made available for instrumentation purposes.

2.1 Applications of Instrumentation

First we would like to summarize research in the field of instrumentation itself.

DTrace [CSL04] is a tool capable of dynamically instrumenting base-level and kernel-level software. Tracing programs are defined in the D language, a subset of C with added functions and variables specific to tracing. DTrace users define probes which are instrumentation points. A probe is composed of a condition and an action. Probes are comparable to pointcuts in aspect-oriented programming. The DTrace framework itself performs no instrumentation of the system; that task is delegated to instrumentation providers. Providers are loadable kernel modules that communicate with the DTrace kernel module. Probes are advertised to consumers, who can enable them by specifying a 4-tuple consisting of an provider, module, function and name, to scope the instrumentation. The provider dynamically instruments the system and the probe's action is executed.

Some researchers have focused on the performance impact that instrumented code might have on program execution. Removing instrumentation once it has fulfilled its purpose is key to reducing the performance impact. This can be achieved by using dynamic instrumentation [HNM+97, SSS00]. Dynamic instrumentation is the capability to dynamically add and remove instrumentations from a running system. Another solution proposed by Arnold and Ryder [AR01] shows that combining instrumentation with sampling leads to accurate profiles (93–98% overlap with a perfect profile) with low overhead (3–6%).

ATOM [SE04] and Purify [HJ92] use instrumentation to collect data about a system. Both these tools use static techniques, instrumentation happening when the analyzed system is not running. ATOM is a framework to build multiple types of program analysis tools. It uses a link-time code modification system. This system builds a symbolic intermediate representation of object files and libraries of a program and the desired analysis routines. Purify detects memory leaks an access errors. This is achieved by inserting instruction checks for every memory read and write by a given system. Afterwards it is able to detect errors like reading uninitialized memory.

2.2 Structural Reflection

Structural reflection enables one to reflect upon the structure of a program. Therefore it grants access to static representations like class and method. This also means that structural reflection is limited to the static representation of programs. McAffer described this as the top-down approach.

Structural reification can be found in many programming languages, like CLOS [BGW93, ABB+89], Java [Sun99], Smalltalk [Riv96] itself or ClassTalk [Coi90] and ObjVLisp [Coi87].

2.3 Behavioral Reflection

In contrast to structural reflection, behavioral reflection deals with execution and is therefore the dynamic part of a system. McAffer described this as the bottom-up approach.

Smith [Smi82, Smi84] pioneered the concept of behavioral reflection in the context of Lisp. He claimed that a reification not related to the structure of the language might be required, for example, for a message send.

Ferber [Fer89] introduced the *message reification* reflective model where each message is an instance of a message class. Each message is responsible for interpreting itself. The message class defines a message send specifying the interpretation. The semantics of message sending can be adapted by subclassing the message class. In Ferber's model the sender of the message was not taken into account in the reification. Cazzola [Caz98] extended this model by including the sender object in the message reification.

2.3.1 CodA

McAffer worked in the realm of distributed objects. McAffer introduced the concept of operational decomposition in which a running application is seen as the set of runtime events required to fulfill its tasks. This approach was introduced in CodA [McA95]. CodA reifies the message send process from an operational decomposition point of view at the meta-level. Conceptually, McAffer suggests to separate the description of the computational behavior of an object from that of its base language. While systems like CLOS and ClassTalk try to extend the functionality of particular language facilities or constructs, CodA deals with constructs that are not language specific.

McAffer observed the need to bring traditional engineering techniques to the meta-level, such as decomposition, abstraction and reuse [McA96]. Up to this point the meta-level has been perceived as a place to perform small changes with no organization at all. Instead, McAffer proposed to see the meta-level as any other complex system built with engineering techniques to achieve a certain meta-level architecture. CodA provides seven meta-object, called meta-level components. In McAffer's view they cover the behaviors essential to common object models.

All meta-level components are a conceptual part of the process of sending a message and can be seen in Figure 2.1. Object *A* sends a Message *M* to Object *B*. Usually this is done directly in Smalltalk. CodA adds a meta-level that segments the message send into six meta-level components:

- *Send* represents the action of an object that represents the delegation or information sharing of that object.
- An *Accept* shows that the message has reached the destination.
- The *Queue* holds all incoming messages until they are processed.
- *Receive* states that the object is now ready to process the next queued message.
- The *Protocol* returns the method of the object that has to be executed.

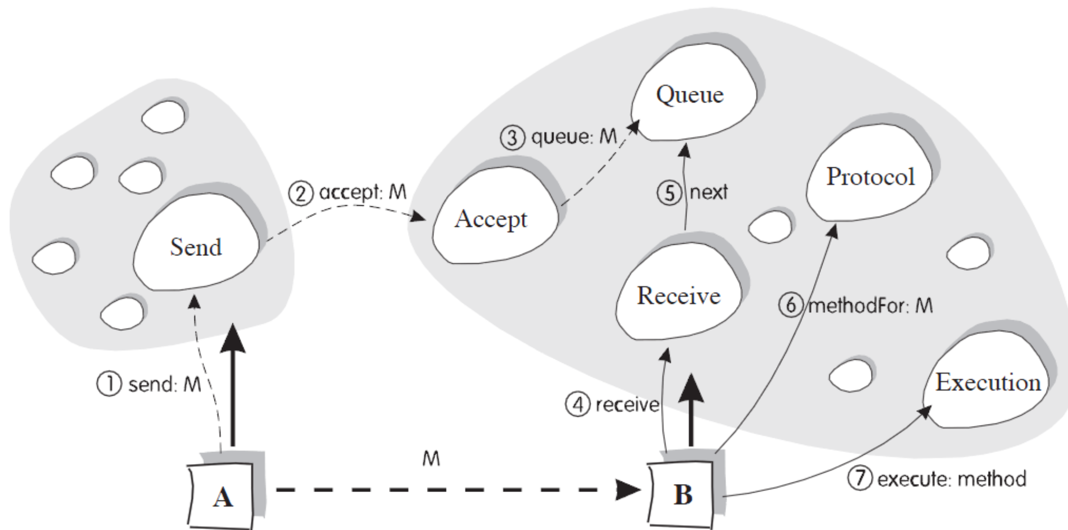

Figure 2.1: CodA reification of message sends

- *Execution* represents the execution of the method, to fulfill the message.

The main drawback of CodA is that it reifies message sends for every message send in the system. This has a negative impact on performance.

2.3.2 Iguana

Iguana [GC96] took the operational decomposition idea literally and reified events which are modeled as reification categories. In contrast to CodA, Iguana offers selective reification, without having to instrument the whole system, or in the case of CodA every message send. This reduces the negative impact on performance and allows one to select program elements down to individual expressions.

Iguana was first developed for C++ and works by placing annotations in the source code to define behavioral reflective actions. It has 23 reification categories. Iguana provides a fine-grained Meta-object protocols (MOPs) [KdRB91, Kic92] which allows different object-models to coexist in a system. Therefore it is possible that the same mechanism, for example object creation, behaves in a different way since different entities, in this case classes, may have different MOPs. If an instance of a class is adapted, no other instance of the class should be affected by this change.

Iguana/J [RC02, RC00] is the implementation of Iguana for Java. Iguana/J was implemented using the Java Just-in Time (JIT) interface. Instead of using annotations in the source code for specifying reflective actions Iguana/J uses a definition file called protocol. This file is compiled by a special Iguana compiler which dynamically generates the code to be executed. This technique is useful since the tool has access to the internal structures of the interpreter. However, this solution is not portable: Sun Microsystems did not continue developing the JIT interface so Iguana/J does not run in the newest VMs. All Iguana implementations have a strong separation of base- and meta-level. The separation can be seen in the Figure 2.2.

The base-level object represents an object of the system we would like to instrument. The meta-object-protocol (named Iguana protocol in the figure) allows the instrumentation of multiple reification-categories

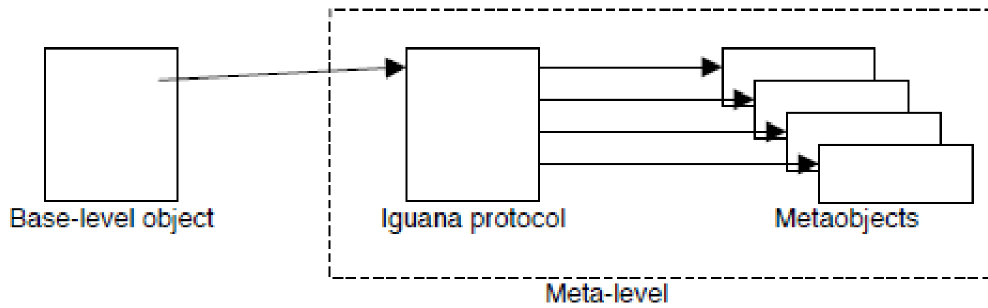

Figure 2.2: Separation of Meta and Base-level in IguanaJ.

at the same time. Reification-categories are the different events Iguana/J implemented which can be reified. IguanaJ implemented the following seven events:

- Object creation, used for classes to add instrumentation whenever a new object of that class is created.
- Object deletion, to reify deletion of a certain object.
- method call, to reify message send.
- method dispatch, to reifiy method selection of a class after a message send.
- method execution, reifies the execution of a method of a certain class or object.
- state read, to reify whenever a certain object's state is read.
- state write, does the same thing as state read for state write.

All seven reification categories together define the canonical events of Iguana/J.

Let us analyze a simple Iguana/J example proposed by Redmond and Cahill [RC02]. In this example the event of a message being executed is reified. Every time that this event reification is triggered at runtime special verbose output before and after the method execution is shown.

```
1 class VerboseExecution extends MExecute {
2   Object execute(Object o, Object[] args, Method m)
3         throws InvocationTargetException,
4               IllegalAccessException {
5     System.out.println("Before method " + m.getName());
6     result = m.invoke(o, args);
7     System.out.println("After method " + m.getName());
8   }
9 }
```

Listing 1: Reification of method execution with Iguana/J

We can observe in this example that the problem domain solution is coupled with the event reification. In lines 1 – 4 the event reification is declared. Lines 5 – 7 contain the problem domain solution. Thus other potential consumers of this reification cannot reuse it. Moreover, they have to duplicate the same reification with their domain specific needs.

When scoping the event reifications, Iguana/J models MOPs that are associated to language constructs. We can see an example in Listing 2 where a `VerboseProtocol` is defined. It is composed of two event reifications: method execution and state write.

```
protocol VerboseProtocol {
  reify Execution: VerboseExecution;
  reify StateWrite: VerboseStateWrite;
}
```

Listing 2: Iguana/J MOPs definition.

In Listing 3 we observe how a protocol is associated with a single object. The Meta abstraction models the object responsible for managing the association between protocols and objects. Protocols can also be associated to classes, in which case they are applied to all instances of the class.

```
MyClass obj = new MyClass();
Meta.associate(obj, "VerboseProtocol");
```

Listing 3: Iguana/J Scoping.

A drawback of this approach is that the event consumer cannot scope dynamically which reifications he wants to react to. This can be seen in Listing 2 of theIguana/J example. The logger—actually not implemented but represented by the print function of the system—is bound to the Verbose Execution. There is no possibility provided to change the focus of the logger from the `VerboseExecution` to another event.

2.4 Partial Behavioral Reflection

Partial reflection was first introduced in the 1990 OOPSLA/ECOOP workshop on Reflection and Meta-level Architectures in Object-Oriented Programming [Ibr91]. *Partial reflection* overcomes inefficiency by making reflective facilities available only where they are needed. This technique avoids the inherent inefficiency of unnecessary reifications arising from *full reflection*. The idea is to make reflective facilities available only when they are needed.

2.4.1 Reflex

A full model of *Partial Behavioral Reflection* was introduced by Tanter *et al.* [TNCC03]. This model is implemented in Reflex for the Java environment. Reflex offers an even more flexible approach than pure Behavioral Reflection. The key advantage is that it provides a means to selectively trigger reflection only when specific, predefined events of interest occur. Reflex uses meta-links to modify the behavior and hooksets to specify where this change should take place. A link invokes messages on a meta-object at occurrences of marked operations. The attributes of a link enable further control of the exact message to be sent to the meta-object. Partial behavioral reflection was first implemented using bytecode transformation in Java. Reflex meta-level engineering is highly flexible however it suffers from a key limitation. Since Reflex is a portable Java extension it works by transforming bytecode. Although the reflective behavior occurs at runtime the framework forces the user to anticipate the reflective needs at load time. This means that Reflex does not allow a programmer to insert new reflective behavior affecting already-loaded classes into a running application. The application has to be stopped, the reflective needs have to be specified and then the application has to be reloaded for the reflective changes to take place.

AspectJ [KLM+97, KHH+01] is an aspect-oriented extension for Java. Tanter points out that there is a major difference between Reflex and AspectJ. With AspectJ the pointcut declaration and the link are embedded within the aspect definition. Although this can be seen as a nice property, it is also an example of coupled behavior: defining a set of points, a behavior, and the link between them are different concerns that should possibly be specified separately. Reflex allows for such separation, which enhances hookset and meta-object reuse.

2.4.2 Reflectivity

Denker introduced Reflectivity [Den08], a Reflex model implementation for Smalltalk. Reflectivity targets three important problems present in the previous tools regarding behavioral reflection. These problems are anticipation, sub-method structure and meta recursion. Iguana/J introduced a working implementation of Unanticipated Partial Behavioral Reflection (UPBR) but suffered from portability issues. Reflex requires the user to anticipate where reflection is going to be needed.

Reflectivity provided UPBR while maintaining portability. This was achieved by using reflective methods that are dynamically compiled thus allowing unanticipated change. Persephone [DDLM07] presented a model for reflective methods and was responsible for recompiling methods that had been reflectively modified.

Reflectivity takes advantage of the reflective structures of Smalltalk. Abstract syntax trees (AST) are used as the sole representation of behavior. The AST is the representation of the source code. Each node represents elements like variables, statements, constants or operators. Reflex hooksets were removed and links were just conceived as annotations to any AST node thus simplifying the Reflex model. Using AST nodes allowed Reflectivity to achieve sub-method reflection capabilities.

Behavioral reflection cannot be applied to the whole system. If the reflection tool modifies basic objects which are used by the tool itself then we end up with infinite recursion (base-level events trigger meta-level events, which trigger further meta-level events, and so on). Reflectivity introduced the concept of *meta-level execution* to reify the level in which the behavior reflection should occur thus avoiding infinite recursion.

Even though the meta-level architecture is as flexible as in Reflex, Reflectivity introduced no enhancements in this area. The link abstraction is highly expressive but to a certain extent is too low level to fully support the meta-level architecture.

2.5 Aspect-oriented Programming

Aspect-oriented Programming (AOP) [Kic96, KLM+97, KIL+97] is a technique that aims at increasing modularity by supporting the separation of cross-cutting concerns. Pointcuts pick out join points, *i.e.,* points in the execution of a program that trigger the execution of additional cross-cutting code called advice. Join points can be defined on the run-time model (*i.e.,* dependent on control flow). Although AOP is used to introduce changes into software systems, the focus is on cross-cutting concerns, rather than on reflecting on the system. Kiczales *et al.* [KLM+97] claim: *"AOP is a goal, for which reflection is one powerful tool."*. Although aspects can be dynamically enabled or disabled, they are specified statically.

The joint points provided by aspect languages are too restrictive and it is not straightforward to extend them: AspectJ's join-point model offers a fixed set of pointcuts, *i.e.,* `call`, `execution`, `static-initialization`, *etc.* Even though these pointcuts can be composed with boolean operators, the user cannot define new pointcuts. This means that the event reifications are fixed to the capacity of this predefined set of pointcuts.

Another drawback is that the consumer has to define the pointcuts and the aspect together. The consumer is coupled to this particular definition of pointcuts. Other consumers that might be interested in the same *events* have to redefine them separately.

Next we will analyze an AOP example for detecting when a figure is moved.

```
1  public abstract aspect MovingFigure {
2
3    pointcut move():
4      call(void FigureElement.setXY(int,int)) ||
5      call(void Point.setX(int))              ||
6      call(void Point.setY(int))              ||
7      call(void Line.setP1(Point))            ||
8      call(void Line.setP2(Point));
9
10   before(): move() {
11     System.out.println("about to move");
12   }
13
14   after() returning: move() {
15     System.out.println("just successfully moved");
16   }
17 }
```

Listing 4: Aspect detecting when a figure object is moved.

In this example[1] we can see the definition of an aspect that defines the events that should occur when a figure is moved. Lines 3–8 define the move pointcut: if any of these methods for these classes is executed then a figure has moved. Next we can observe two advices, one just before the figure moves and the other after the figure has moved.

In this way these events (pointcuts) are coupled to the behavior defined in the advice. These events are thus not reusable. If another aspect would like to profit from the definition of this advice it cannot reuse this instrumentation. The new aspect will have to redefine the same pointcut and its aspects.

2.6 Aspect-oriented- with Component-based Programming

Suvée, Vanderperren and Jonckers introduced JAsCo [SVJ03], an aspect oriented implementation language in Java. The basic idea of JAsCo is to bring component based software development (CBSD) to aspect oriented software development (AOSD). The goal of CBSD is to separate the software into multiple independent components (black boxes) to augment modularity, reusability and the speed of development.

JAsCo introduces two new concepts: Aspect Beans and connectors. Aspect Beans are basically Java beans—reusable software components conforming to a particular convention—with one more hook. These hooks can be seen as a combination of AspectJ's pointcut and advice. The main difference is that aspect beans are reusable. The architecture in Figure 2.3 shows the bean comp1 on the left side, which will defer execution to the connector registry once a method is called. In turn every connector then dispatches to the hooks. Furthermore there are connectors. The responsibility of a connector is the initialization of hooks with a specific context. All aspect beans are registered in the connector registry with informs the connectors when a aspect bean is executed. Connector can map to several hooks that represent the additional functionality like logging or other cross-cutting concerns.

The implementation of aspect beans allows the aspects to be reused. But while connectors can be loaded and unloaded at runtime, aspects may not. Only existing events or methods can be used in aspect beans. Therefore there is no way of defining a new event.

[1]http://www.eclipse.org/aspectj/doc/released/progguide/starting-aspectj.html

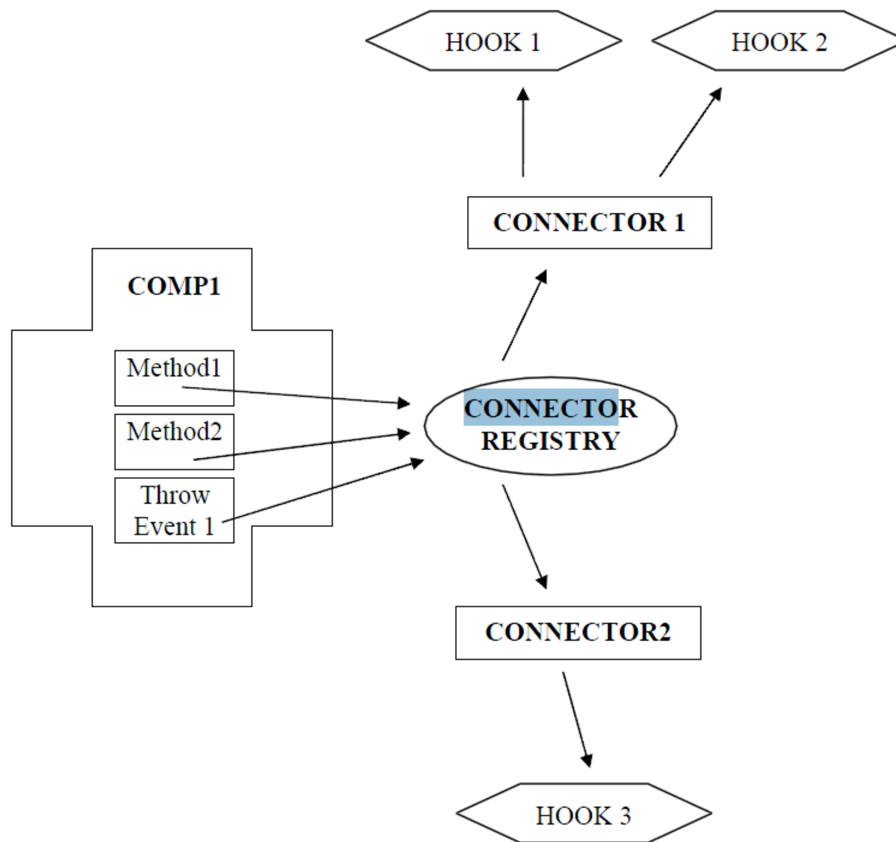

Figure 2.3: JAsCo architecture.

2.7 Event-based Aspect-oriented Programming

Douence, Motelet and Südholt [DMS01] introduced a general operational model for crosscutting based on execution monitors called Event-based Aspect-oriented Programming (EAOP). They proposed a formal model for the definition and detection of events patterns. They describe an event as the representation of a point in the program execution. In their prototype they implemented the explicit events method call and method return.

Douence and Südholt [DS02] later introduced constructor calls and constructor returns as events. The Execution Monitor in their implementation observes events emitted during execution. The execution of the base program is suspended when an event is emitted. The monitor matches this event against different event patterns. When a pattern is satisfied the associated actions are executed.

The event is then propagated to all aspects. After each aspect in turn has reacted to the event the control is given back to the base program.

A key drawback of EAOP is that the event/pointcut definition is coupled to the consumer behavior thus the event abstraction is not reusable. This drawback is also present in pure AOP approaches.

Another important drawback is that this approach provides only four events: constructor call and return, and method call and return. Although the authors claim that it is possible to extend this set of events with state read and write they do not describe a solution to this. Neither do they provide a mechanism for developers to generate custom events. Moreover the reifications that are introduced in each of these events

are fixed.

We can summarize the key issues with existing approaches to instrumentation as follows:

Domain-Polluted Instrumentation. Most approaches pollute the instrumentation with domain behavior thus rendering this instrumentation only usable for a particular consumer or domain. Instrumentation behavior and its consumers are tightly coupled.

Language-biased events. Events are modeled after language constructs and in many cases it is not clear how to extend the basic set of events.

Instrumentation-oriented scoping. Existing approaches only provide a single scoping mechanism which is coupled to the instrumentation.

3
Iguana/S

IguanaS was our first approach. Our general idea was to use behavioral reflection to implement instrumentation. We chose behavioral reflection because we see the system like McAffer, as "a set of events happening to achieve a certain goal". Also in contrast to structural reflection, behavioral reflection is language independent. We took IguanaJ as our starting point for the model. To keep true to the idea of McAffer we wanted to model explicit events in the system.

Another goal to bring back events to Unanticipated Behavioral Reflection. Therefore the idea was to mix the events of Iguana with unanticipated behavioral reflection of reflectivity.

The Iguana system consists of seven reification-categories representing seven events. Reflectivity on the other hand implements links to change AST and does not model events. But Reflectivity has another advantage. It still uses links like Reflex but they are attached to AST nodes and not to byte-code.

3.1 Implementation

IguanaS is built on top of Albedo [RRGN10]. AST nodes are used in Albedo to represent behavior. Additional behavior can be added with links. A link simply points to a Meta-Object. The Meta-Object specifies the additional functionality. If an AST node is linked to a meta-object and is executed, the AST-node will be recompiled and its behavior changes to the one specified in the Meta-Object. It might add new functionality, replace or prevent existing functionality.

We implemented the Iguana reification categories as meta-objects. Therefore we implemented Meta-Objects for method execution, object creation and deletion and state read and write. They will be explained in detail in the next subsections. Furthermore we implemented an instrumentor called `Meta`. `Meta` links the Meta-Objects with the corresponding AST node of a desired Object. `Meta` is also responsible for any necessary preparations in the target object, like creating a `new` method to instrument object creation.

Method Execution

To be able to track method execution, the method has to be annotated. A method is annotated by getting the `MethodNode` of the chosen method of the corresponding class and adding the link to it. The `MethodNode` is obtained by searching the `MethodDictionary` of a Class for the desired method, which is stored as a

`CompiledMethod`. Then you have to convert it to a `MethodNode` with `CompiledMethod>>methodNode`.
In this implementation we annotate all Methods of a Class, but it is possible to implement method execution only for a specific Method.

Object Creation

Object creation is detected by annotating the `new` method of the corresponding class. But the method can be missing in a class. Therefore the `new` method is added with an usual body: `return self basicNew` This is done using `ClassDescription>>compile:classified:notifying:`. After that, we only have to add it to the `MethodDictionarry` with its name as a `Symbol`. Afterwards the link can be added to this (or the already existing) method, the same way as in method execution.

Object Deletion

To be able to detect if an object is garbage collected we had to add the object to a `WeakRegistry`. If an Object is saved in a `WeakRegistry`, it will not count as reference. But if all other references to the object are lost, the garbage collector will collect the object. This happens after the `finalize` method of the object is executed. Then the object will be collected by the garbage collector and is no more referenced in the weak registry. Therefore, if not already existing, a `finalize` method has to be added to the object we want to reify. This is done in the same way as in object creation, but with an empty body. Afterwards the `link` is added to this method.

State write

Because there are no slots in Smalltalk, `Assignment Nodes` have to be annotated to be able to track state write. They are found nested inside the methodNodes. Because there can be a high amount of nesting, visiting each node is the recommended approach. By subclassing the `RBProgramNodeVisitor` and overwriting `RBProgramNodeVisitor>>acceptAssignmentNode` we can reify the detection to check if the assignment node has already been annotated and if the assignment happens to an instance variable after all.

State read

For state read `VariableNodes` needs to be annotated. We overwrite the `RBProgramNodeVisitor>> acceptVariableNode`. Again we have to check if the variable is an Instance and if this annotation has already been added.
Because both state read and write use the same Visitors we had to add a boolean test to be sure, that assignment nodes are only annotated when a state write is reified. The other way we have to check if it is a state read reification to add a link to a variable node.

Protocol

In IguanaS a protocol is a meta-object-protocol similar to Iguana/J. It allows the instrumentation of multiple events at the same time and therefore eases the annotation of events. With protocol it's only necessary to declare which metaobject has to be added to which class. Iguana/J implemented a special compiler to interpret the protocol and execute the instructions through the JIT-Interface. Without protocol you can not instrument in Iguana/J. This is different in IguanaS, where you are able to instrument events one at a time without the use for protocol. But by using protocols you are able to add multiple protocols to one Object or class in IguanaS, which is not possible in Iguana/J.

3.2 Findings

After implementing Iguana/S we arrived at the following conclusions:

- Although Brendan Gowing, Vinny Cahill and Barry Redmond, the creators of Iguana respectively Iguana/J, talk about events and implementing event-like reification-categories, the Iguana system does not have explicit events that are triggered when an actual event happens. Instead Iguana and IguanaJ have implicit events that are triggered by the reification categories. This implies that there is a gap between the abstraction of McAffer who talks about explicit events and the model of Iguana with its reification-categories. We want to make events explicit.
- Iguana defines all its events as canonical. The reason is that, unlike Smalltalk coda, Java cannot represent uniformly all interactions as message sends. Therefore the events in Iguana are language biased. An interesting new goal would be to find a smaller set of canonical events, if possible a single one.
- AST-adaptation is at a higher level of abstraction compared to the Iguana byte-code manipulation. AST nodes are used in a lot of programming languages, thus making our solution more portable and language independent, like Reflectivity. Because of those reasons we wanted to keep AST nodes as the representation of events.

4 Chameleon

Our goal with the prototype, is an implementation of the ideas we had by being influenced by previous work and the findings of the previous section. We propose to solve these shortcomings by means of explicit meta-events. In this section we introduce Chameleon[1], a Smalltalk [GR83] prototype of our approach.

Chameleon models meta-events explicitly and separates the specific behavior of a development tool from the instrumentation by applying the observer pattern. Chameleon provides a simplified event architecture with a single canonical event on top of which any other meta-level event can be defined. This is achieved by integrating two key approaches to reflection: CodA/Iguana's event-oriented approach and Reflex/Reflectivity's partial behavioral reflection approach.

Events are the building blocks of both CodA and Iguana, however, in neither framework are they modeled explicitly. By explicitly modeling events and applying the observer pattern, a better separation of concerns can be achieved.

Figure 4.1 shows a class diagram of Chameleon's key abstractions which we will discuss next.

4.1 Events

Chameleon offers a single canonical event which models the execution of a single AST node. On top of the AST node execution event every other object-related event can be built. Also, by having explicit events, we can enforce developer tools to use the observer pattern to listen to these meta-events. Thus, a better separation of concerns between event generation and domain requirements is achieved.

The class of an event models the conceptual abstraction of that event. Each occurrence of an event is modeled with a new instance created from the event class. The responsibilities of an event class are:

- Determine where the event should be reified and signaled. The event class knows which AST nodes, when executed, should reify the event.
- Describe which dynamic data is required to reify the event. For example, a message send event contains and reifies the sender object, the receiver object and arguments of the message.

[1] `http://scg.unibe.ch/research/chameleon/`

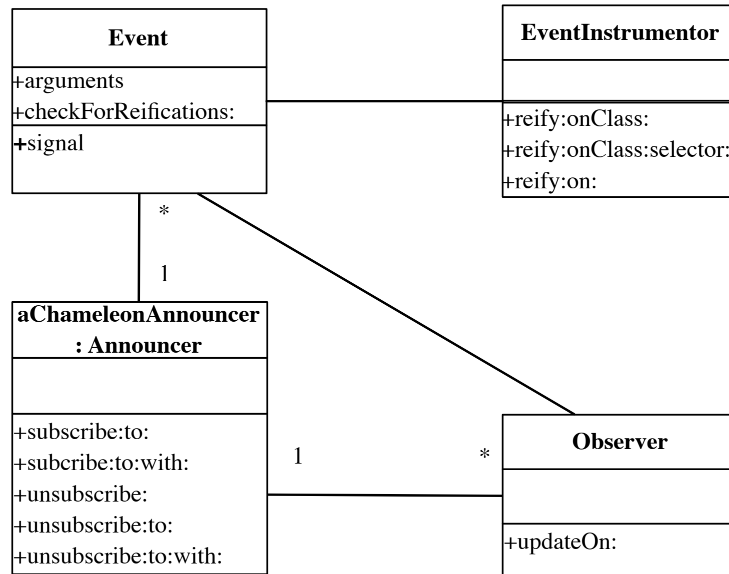

Figure 4.1: Chameleon's core abstractions.

- Knowing what adaptation has to take place on an object so the event can be reified. For example, when reifying an object creation event it might happen that the message `new` is not overridden in the instrumented object. Because of this, the instrumentation tool should add the `new` method so the event can be reified.

Every time an event is triggered a new instance of this event is created. An event instance is responsible for knowing how to signal itself. Every event instance holds extra reifications which depend on the event, for example, sender of a method, variable written.

We have implemented the Iguana/J canonical events to show the capacities of Chameleon approach. These events are: object creation, object deletion, method execution, message send, message dispatch, state write and state read. For a detailed explanation of these events look at section 6.3.

4.2 Instrumentation for Signaling Events

The `EventInstrumentor` is responsible for instrumenting an application to reify specific events under specific circumstances.

Instrumentation is performed at the AST level. The `EventInstrumentor` has the following responsibilities:

- Instrument AST nodes to reify an event. The instrumentation has two responsibilities: create the event and signal it through the announcer.
- Provide different scopes for instrumentation. For example, an event can be reified for several classes or a single class, or for a single method or for a single node.
- Adapt the application to allow an event to be reified. In the case of the `ObjectCreationEvent` the `new` method has to be added before the reification instrumentation could take place.

Chameleon provides different scopes when instrumenting an event. Each event can be reified either on a class, a method or a single node. As an example we reify method execution on a bank account class.[2]

```
(EventInstrumentor new) reify: MethodExecutionEvent onClass: BankAccount
```

Listing 5: Reification of method execution inside a class

The method `EventInstrumentor>>reify:onClass:` handles the instrumentation. Another possible scope is the method:

```
(EventInstrumentor new) reify: MessageSendEvent onClass: BankAccount selector:#statement
```

Listing 6: Reification of message send inside a method

In this example we reify the message send event on the statement method of the bank-account class.

The third scope can be defined for a single node:

```
(EventInstrumentor new) reify: ASTNodeExecutionEvent on: aNode
```

Listing 7: Reification of node execution on a single node

Here we reify the node execution on a node.

4.3 Announcer

The responsibility of the `Announcer` is to provide the observers with the possibility to subscribe and unsubscribe to events. The `Announcer` provides different scopes for the subscription. An observer can subscribe to all events or all occurrences of a particular event. In the listing below we can see a profiler subscribing to all events:

```
aChameleonAnnouncer
   subscribe: aProfiler
```

Listing 8: Profiler subscribing to all events.

For simplicity reasons a global instrumentor which uses a global announcer is defined. However, the design does not force the users to only use these global objects. Developers are free to instantiate new instrumentors and announcers creating contextual event refication environments.

Next, we can see a profiler subscribing to a method execution event.

```
aChameleonAnnouncer
   subscribe: aProfiler
   to: MethodExecutionEvent
```

Listing 9: Profiler subscribing to the method execution event.

To inform the announcer about an event execution, the initialization of a event has to call the `Announcer>>announce: anEvent` method.

An observing customer is able to unsubscribe from either all events or an event type as seen in the following listings:

[2] Readers unfamiliar with the syntax of Smalltalk might want to read the code examples aloud and interpret them as normal sentences: An invocation of a method named `method:with:`, using two arguments looks like: `receiver method: arg1 with: arg2`. Other syntactic elements of Smalltalk are: the dot to separate statements: `statement1. statement2`; square brackets to denote code blocks or anonymous functions: `[ statements ]`; and single quotes to delimit strings: `'a string'`. The caret ^ returns the result of the following expression.

```
aChameleonAnnouncer
   unsubscribe: aProfiler
```

Listing 10: Profiler unsubscribing to all events.

```
aChameleonAnnouncer
   unsubscribe: aProfiler
   to: MethodExecutionEvent
```

Listing 11: Profiler unsubscribing to the method execution event.

4.4 Observers

An observer models different development tools like profilers, code coverage analysis and loggers. Observers subscribe through the `Announcer` to listen to specific events. The particularities of the domain are kept in the observer thus providing good separation of concern between the instrumentation and the problem domain. The observer do not have to follow any particular pattern but the observer pattern to listen to event and act accordingly.

Observers are notified by the announcer when an event has been signaled and reified with the message `updateOn: anEvents`.

5 Validation

In this section we will demonstrate how Chameleon resolves the three drawbacks of existing instrumentation approaches.

5.1 Domain-Polluted Instrumentation

MetaSpy [BNRR11] is a framework to build domain specific profilers. MetaSpy offers abstractions to model profilers and instrumentors. A profiler uses various instrumentors to reify domain specific events.

In a first case study of MetaSpy, the authors built a domain specific profiler for Mondrian [MGL06], a software visualization tool. The default visualization displays a software system as nodes and edges representing classes and inheritance. Mondrian can be customized to change what the nodes and edges represent.

The authors of MetaSpy built a domain-specific profiler for Mondrian to detect the source of a certain performance issue. Each time Mondrian detects a change in a node it refreshes the whole visualization. This profiler's goal was to measure the number of times the `displayOn:` method was called for each of the nodes in a Mondrian visualization.

The MetaSpy domain-specific profiler was defined as follows:

```
1  MondrianProfiler>>setUp
2     self model root allNodes do: [ :node |
3        self
4           observeObject: node
5           selector: #displayOn:
6           do: [ :receiver :selector :arguments |
7              actualCounter
8                 at: receiver
9                 put: ((actualCounter
10                      at: receiver
11                      ifAbsent: [ 0 ]) + 1) ] ]
```

Listing 12: Attaching the profiler to the Mondrian-nodes

This profiler defines that every node in the model will be observed for invocations of the `displayOn:` method. The block from lines 6–11 defines the action that should be executed when the message `displayOn:` is received by any node. The block counts for each node how many times the method is executed.

The message `MetaSpy>>observeObject:selector:do:` delegates to an instrumentor which is responsible for performing the instrumentation of the nodes.

This process of instrumentation is described by the MetaSpy authors as follows:

> *An instrumentation strategy is responsible for adapting a domain-specific model and triggering specific actions in the profiler when certain events occur.*

This statement indicates coupling between the profiler and the instrumentation. The MetaSpy instrumentor introduces the profiler domain-specific behavior in the nodes where the reification of interesting events should happen. This reification cannot be reused since it is coupled to this particular domain. We call this domain-polluted instrumentation, because the instrumentation is coupled with the profiler.

If we would add a second MetaSpy profiler the same event has to be reified again. This is due to the fact that each instrumentation hooks the functionality directly into the code. So every newly added profiler adds a new instrumentation and therefore additional code (besides the profiler) to the system. Therefore the coupling of instrumentation and the profiler has not only the drawback of not being able to reuse the instrumentation event reifications but it also has performance impact. Adding multiple profilers that use the same instrumentation adds the same reification for each profiler to the code and thus slowing the system unnecessarily.

Chameleon on the other hand separates the instrumentation reification from the development tools. In the case-study of Mondrian we would need a method execution event to have the correct representation of the `displayOn:` method execution.

```
(Instrumentor new)
   reify: MethodExecutionEvent
   onClass: Node
   selector: #displayOn:
```

Listing 13: Reifying method execution for the method displayOn: in the class node of Mondrian

The Profiler only has to register for this event:

```
aChameleonAnnouncer
   subscribe: mondrianProfiler
   to: MethodExecutionEvent
```

Listing 14: Profiler registers for the method execution event.

If another development tool wants to observe this method execution, we just have to register it, since the instrumentation is already there. This is shown in the code below:

```
aChameleonAnnouncer
   subscribe: logger
   to: MethodExecutionEvent
```

Listing 15: Logger registers for the method execution event

The event does not need to be reified again. Therefore we avoid domain-polluted instrumentation.

The use of explicit events makes the reuse of instrumentation possible and provides the capability to always know what is already instrumented. It also reduces the performance impact whenever multiple development tools use the same instrumentation.

5.2 Language-biased Events

Lienhard *et al.* [LGN08] proposed an object-oriented back-in-time debugger. This kind of debugger is extremely useful for identifying the causes of bugs that corrupt execution state without immediately raising an error, as they allow us to inspect the past states of objects no longer present in the current execution stack. To remember the flow of objects this debugger has to answer a key question: How was this object passed here? This means that for any object accessible in the debugger, the tool has to be able to inspect all origins up until the allocation of the object. This also allows us to find out where a particular value of a variable comes from.

The approach of Lienhard *et al.* was to modify the Smalltalk virtual machine at key points to gather the state of variables and instance variable written and read. This also required to take into account the values of the arguments of method invocations.

Let us analyze the possibility of implementing a similar solution using Iguana/J. Message invocations as well as state read and write are provided as canonical events. State read and write events only model the accesses to instance variables but not regular variables. Specific behavior for gathering the required data can be introduced for these events. However, there is a key event that we cannot model with Iguana/J, when a value is assigned to a simple variable. It is not possible to reify this event with the provided canonical events. Iguana/J canonical events are implemented by modifying the Java VM by using the JIT interface. Thus, a variable assignment event would require modifying the Java VM to introduce the new assignment event. In Iguana/J each of the seven canonical events matches a particular virtual machine adaptation. Moreover, some events like object creation are required since in Java an object is created through constructors instead of normal messages to the class.

Our approach proposes an unbiased event implementation by building on top of the AST abstractions. This means that events match AST nodes with particular characteristics. We can find AST abstractions in many different languages, therefore our framework can be ported to other languages.

To build the variable assignment event we need to extend `ASTNodeExecutionEvent` and override the method `shouldBeReifiedOn:`.

```
VariableAssignmentEvent>>shouldBeReifiedOn: aNode
   ˆ aNode isAssignment
      and: [ aNode variable isVariable ]
```

Listing 16: Selection of assignment nodes for reifying the event that a variable was assigned.

Only the nodes that are assignments and whose variable side is a variable and not a field will be selected.

5.3 Static Instrumentation Scoping

Bockisch *et al.* proposed a new way to define events which can be composed and concatenated. The next snippet of code describes their motivating example of a shopping cart discount. Every time a low demanded product is purchased a discount is applied to the purchase value.

There is an explicit definition for a `LowActivity` event which models that this event should be triggered when the product has low demand. `LowActivityPurchase` is triggered after an `LowActivity` event. When a purchase is made and the product involved has low demand then a `LowActivityPurchase` event is triggered. If these two events are sequentially triggered the `LowActivityDiscount` aspect is executed applying a discount.

```
1  event LowActivity(P product){
2    int LOWER_BOUND = 100;
3    Info purchaseInfo = new Info();
4    after(Purchase purchase): RelevantPurchase(purchase) {
5    purchaseInfo.increase(purchase.product());
6    }
7    when(P product): call(P.timeDone()) && target(product) {
8      if (purchaseInfo.count(product) < LOWER_BOUND) {
9        trigger(product);
10     }
11   purchaseInfo.reset(product);
12   }
13 }
14
15 event LowActivityPurchase(C cart) {
16   Set<P>lowActivityProducts = new SetyP>();
17   after(P product): LowActivity(product) {
18     lowActivityProducts.add(product);
19   }
20   when(Purchase purchase): RelevantPurchase(purchase) {
21   if (lowActivityProducts.contains(purchase.product())) {
22     trigger(purchase.cart());
23   }
24 }
25
26 aspect LowActivtyDiscount {
27   before(C cart): LowActivityPurchase(cart) {
28     cart.applydiscount(10);
29   }
30 }
```

Listing 17: Bockisch *et al.* event declaration for a low activity discount aspect.

To produce the same results in Chameleon we need to trigger an event when a purchase is made. We assume that there is a purchase method defined for class `Cart`. The goal is to produce an event each time the purchase method is executed. The event `PurchaseEvent` inherits from `MethodExecutionEvent`. So far, no additional condition is defined. The next snippet of code shows the instrumentation of the method `Cart>>purchase`:

```
(EventInstrumentor new) reify: PurchaseEvent
  onClass: Cart selector #purchase
```

Listing 18: Reification of a purchase event.

Afterwards, a `DiscountChecker` subscribes to the newly installed event.

```
1 aChameleonAnnouncer
2   subscribe: aDiscountChecker
3   to: PurchaseEvent
```

Listing 19: Subscription to purchase event.

A discount is applied when the event is triggered and the `DiscountChecker` determines that the product involved is eligible for discounts.

So both approaches can deal with this general situation, in which the system is instrumented to check if a discount should be applied to a purchase.

Let us assume that we would like to add a customer benefit system based on purchase points. For every purchase that a customer makes he gets a certain number of points which can be used for future purchases. This benefits point program is optional. The approach of Bockisch *et al.* would solve this new requirement

by adding a new event and aspect. In line 1–6 the event models when a customer with a points program makes a purchase. In line 8–12 the aspect keeps the accounting of the customer points depending on the purchase.

```
event PointsPurchase(C cart) {
  when(Purchase purchase): RelevantPurchase(purchase) {
  if (purchase.customer().hasPointSystem) {
    trigger(purchase.cart());
  }
}

aspect CalculatePoints {
  before(C cart): PointsPurchase(cart) {
    cart.applypoints();
  }
}
```

Listing 20: Aspect solution for a customer points system.

Let us assume that the whole system is now running. There is one customer who did not pay his bills in time. To prevent him from getting more customer points the system should temporarily exclude him from gaining more points. This minor change would force the Bockisch *et al.* approach to change the event's conditions to check for unpaid bills. Every time that there is a constraint change we need to modify by hand the conditions in the events.

Our approach avoids this situation by allowing the user to cherry-pick which objects should produce the events. In this case, we only need to detect when a customer does not pay a bill on time and then remove the instrumentation from his cart. Next time, this particular customer makes a purchase the points system is not triggered.

Chameleon also needs to define a new event modeling when a customer with a points program makes a purchase:

```
(EventInstrumentor new) reify: PointsPurchaseEvent
  onObject: aCart selector: purchase
```

Listing 21: Chameleon purchase of a customer with a points program.

With this dynamic instrumentation scoping technique we can dynamically control the scoping of which object should trigger which events, thus preventing the need to build complicated conditions on the events.

Another issue solved by the approach of Bockisch *et al.* is controlling whether aspects should be applied to other aspects too. When two or more aspects are woven into a system, and both aspects profile the system, it is not clear if they have to profile each other too. Every AOP approach has to confront this problem. In the approach of Bockisch *et al.* the so-called development practices are used to resolve those entanglements correctly.

```
aspect DevelopmentPractices composes Logger, Proactive, Prevention {
  local declare precedence Logger, Proactive;
  local declare precedence Logger, Prevention;
  local declare overriding Proactive, Prevention;
  local declare ignoring Logger, Proactive;
}
```

Listing 22: Development Practice for Logger, Proactive and Prevention

In Chameleon this precedence definition is not required. Which portions of the running system are instrumented to produce events is controlled by applying these instrumentations on top of specific objects. Since the instrumentation, observers and events are objects too, in the presence of a new event we can chose which objects should be instrumented. We think in terms of a running system composed of objects.

6 Implementation

Chameleon is built on top of the Bifröst reflection framework [RRGN10]. Bifröst offers fine-grained unanticipated dynamic structural and behavioral reflection through meta-objects.

6.1 Managing AST Meta-Objects

The `EventInstrumentor` is responsible in Chameleon for managing the Bifröst AST meta-objects. An AST meta-object is responsible for adapting the compilation process. These meta-objects are bound to AST nodes which when compiled to introduce extra behavior in the method. When this method is executed the adapted version is run. These meta-objects are transparently managed. The `EventInstrumentor` attaches AST meta-objects to AST nodes to reify different events.

When a node with a meta-object is executed, the meta-object will generate a new event. In our example it is a `MethodExecutionEvent`. The corresponding meta-object creation is seen below:

```
1 ASTMetaObject new
2       delegatingTo: anEvent reificationBlock;
3       arguments: anEvent arguments.
```

Listing 23: Bifröst AST meta-object for event reification.

We can see in Listing 23 the definition of an AST meta-object. When an AST node is executed and this meta-object is attached to it then the block in line 2 is evaluated with the events arguments in line 3. The event is responsible for providing a reification block which defines how the event is reified and the arguments this block should reify too. Then the event reification method for a single node in the `EventInstrumentor` is defined as follows:

```
1 EventInstrumentor>>reify: anEventClass on: aNode
2    | metaObject |
3    metaObject := ASTMetaObject new
4       delegatingTo: anEventClass reificationBlock;
5       arguments: anEventClass arguments
6    aNode metaObject: metaObject.
```

Listing 24: Event reification method for a single node.

In line 6 we are associate the AST node to the meta-object.

As an example, let us analyze the `MethodExecutionEvent`.

```
MethodExecutionEvent class>>reificationBlock
   ^ [ :selector :class :arguments |
      MethodExecutionEvent
         signalMethod: selector
         class: class
         arguments: arguments]
```

Listing 25: Reification block for reifying the execution of an event.

```
MethodExecutionEvent class>>arguments
   ^ #(selector class arguments)
```

Listing 26: Arguments for the reification block for reifying the execution of an event.

We can observe in Listing 25 that the reification block only signals the event with parameters defined by `arguments`. The name of the executed method, the class and the arguments are reified together with the event. Every event defines different reification blocks and arguments.

6.2 Instrumentation Details

The instrumentation of an event requires several steps. Chameleon provides the behavior to reify an event on all methods of a class.

```
EventInstrumentor>>reify: anEventClass onClass: aClass
   | allNodes reificationNodes |
   self targetClass: aClass.
   self event: anEvent.
   self event adapt: self.
   allNodes := Set new.
   (aClass methodDict keys
      do: [ :key | (aClass>>key) parseTree allChildren
         do: [ :node | allNodes add: node ] ] ).
   reificationNodes := anEventClass
                        reificationNodesIn: allNodes.
   reificationNodes do: [ :node |
                     self reify: anEventClass on: node ].
```

Listing 27: Reifying an event for all the methods of a class.

First, the instrumentor needs to find all nodes for all methods of a class. The instrumentor iterates over all methods to obtain all the child nodes including the method node. The decision of which nodes reify the event is delegated to the event itself with the message `reificationNodesIn: nodes`. This message answers a set of nodes that reify the event.

```
ASTNodeExecutionEvent>>reificationNodesIn: aSet
 ^ aSet select: [ :node | self shouldBeReifiedOn: node ]
```

Listing 28: Event delegation to decide which AST node reifies an specific event.

In Listing 28 we can observe the default implementation of `reificationNodesIn:`. In this method the decision whether a node reifies an event or not is delegated to the event itself through the method `shouldBeReifiedOn: node`.

In the case of the `MethodExectuionEvent` the implementation of `shouldBeReifiedOn:` states that any node that is an AST method node should reify the event that a method is being executed.

```
MethodExectuionEvent>>shouldBeReifiedOn: aNode
   ^ aNode isMethod
```

Listing 29: Selection of method nodes for reifying the event that a method was executed.

In the case of a `ASTNodeExecutionEvent` every node inside the chosen scope will be instrumented. For all other events there are different conditions. For `MessageSendEvent` only message send nodes reify this event. `StateReadEvent` is reified by nodes which are variable nodes whose name is also a class instance variable name and are not part of an assignment.

Once the nodes that should reify an specific event are identified the `EventInstrumentor` applies AST node instrumentation on them. This instrumentation adds the necessary behavior to trigger the reified event every time each of these nodes are executed.

Now all method nodes of the bank-account class are reified with method execution events. Every time a bank account receives a message and then the method is executed the `MethodExecutionEvent` will be signaled.

Chameleon also provides behavior for instrumenting specific methods in certain classes.

```
1  EventInstrumentor>>reify: anEventClass onClass: aClass selector: aKey
2     | allNodes reificationNodes |
3     self targetClass: aClass.
4     anEventClasst adapt: self.
5     allNodes := (aClass>>aKey) parseTree allChildren
6                    asSet.
7     reificationNodes := anEventClass
8                    reificationNodesIn: allNodes.
9     reificationNodes do: [ :node |
10                   self reify: anEventClass On: node].
```

Listing 30: Reifying an event for a particular method of a class.

Here only all nodes within this particular method are reified.

For example, in the case of the `ObjectCreationEvent` the node on which the event should be reified might not exist. The object creation event depends on the existence of the `new` method. Generally this method is not overridden and is inherited from the superclass. Therefore the method execution event needs the new method to be present in the class. The instrumentor allows the event to adapt the application to add the required node for the reification. This is done through the `adapt:` method which double dispatch through the instrumentor to perform the right adaptation.

```
1  ObjectCreationEvent>>adapt: anEventInstrumentor
2     anEventInstrumentor
3        addMethod: 'new
4        ^ self basicNew.'
5        selector: #new
```

Listing 31: Application adaptation for the reification of the method execution event.

This creates a new method (if not already existing) in the target class.

6.3 Extending Events

We demonstrate that this approach is more general by implementing Iguana/J's canonical events on top of Chameleon. Therefore we implemented `MessageSendEvent`, `MethodExecutionEvent`, `ObjectCreationEvent`, `ObjectDeletionEvent`, `StateReadEvent` and `StateWriteEvent` on top of `AstNodeExecutionEvent`.

The method dispatch event was not modeled since it can be easily reified using the message send event. Note that the event `MessageReceiveEvent` was also implemented even though it is not part of Iguana/J canonical events. `MessageReceiveEvent` accepts the same nodes as message send but returns different arguments for the observers. It signals the node, receiver and the message. This event is particularly useful when modeling message meta-level management like CodA.

6.4 Benchmark

Using meta-level programming techniques on a runtime system can have a significant performance impact. Consider a benchmark in which a test method is invoked one million times from within a loop. We measure the execution time of the benchmark with Bifröst reifying the 10^6 method activations of the test method. This shows that in the reflective case the code runs about 35 times slower than in the non-reified one. However, for a real-world application with only few reifications the performance impact is not significant when compared to the uninstrumented application. Bifröst's meta-objects provide a way of adapting selected objects thus allowing reflection to be applied within a fine-grained scope only. This provides a natural way of controlling the performance impact of reflective changes.

Let us reflect on the Mondrian use case presented in section 5.1. The main source of performance degradation is from the execution of the method `displayOn:` and thus whenever a node is redisplayed. We developed a benchmark where the user interaction with the Mondrian easel is simulated to avoid human delay pollution in the exercise. In this benchmark we redraw one thousand times the nodes in the Mondrian visualization. This implies that the method `displayOn:` is called extensively. The results showed that the profiler-oriented instrumentation produces on average a 20% performance impact. The user of this Mondrian visualization can hardly detect the delay in the drawing process. Note that our implementation has not been aggressively optimized.

7 Conclusion

In this thesis we have presented Chameleon, a prototype modeling the meta-level as explicit meta-events observed by development tools. Chameleon realizes a strict separation of concerns between instrumentation and the consumers of events, thus resolving the problem of domain polluted instrumentation. Furthermore it assures that events and development tools can be reused. Moreover, we have presented a simplified approach to behavioral reflection through operational decomposition. Our approach proposes a single canonical event—AST node execution—on top of which any other meta-level event reification can be defined. Furthermore our design allows one to add new events by following simple steps. By explicitly modeling meta-events the scoping of the development tools can happen at instrumentation time or at event reification time.

The three presented main drawbacks of previous instrumentation approaches have been solved in Chameleon in the following matter:

7.1 Separation of Concerns

Existing approaches to instrumentation fail to fully achieve separation of concerns as manifested by domain-polluted instrumentation: instrumentation is too tightly coupled to the needs of a particular tool to permit reuse of instrumented events for multiple purposes. Chameleon resolves this by modeling instrumented events explicitly and requiring clients to subscribe to these events by applying the observer pattern. Every event defines what information should be reified in an announcement. The message send event, for example reifies the node, receiver, the message and possible arguments. Any observing development tool is able to listen to any event and use the information in the signaling for any kind of evaluation, for example logging, code coverage and profiling. More tools requiring these meta-events can register to listen to them and reuse the already installed event reification, thus not requiring new instrumentation of the application. Furthermore the same event can be followed by multiple instrumentation objects without the need to change the event or the signaling.

7.2 Dynamic Observer-oriented Scoping

There are various options for a development tool to scope the events to listen to. The development tool can listen to a particular event type, for example message send, thus listening to all message send events occurring in the system. This tool can apply a condition on the events that is listening to to rule out certain events that are not of any interest to the tool domain.

From the instrumentation point of view, particular events with specific triggering conditions can also be installed, for example, a message send which is only triggered for a particular message name. A new event type can be defined and the development tool is registered as an observer of this event.

Furthermore an observer is able to plug and unplug instrumentation for specific objects. This eliminates the need to add conditions on event reification.

7.3 Unbiased Events

A key point in Chameleon is its capacity to extend the single canonical event. Extending means to provide the means to add user-defined events to the system following the requirements of the development tools domains. The following steps are necessary for this:

1. Create a subclass of the canonical `ASTNodeExecutionEvent` or any other predefined event.
2. Afterwards the newly created event must define which kind of AST nodes, when executed, should reify this particular event. For this purpose the method `shouldBeReifiedOn:` needs to be overridden.
3. If this node does not exist then the necessary adaptation behavior needs to be defined by overriding the method `adapt:`. There you can tell the instrumentor to add another method with using `EventInstrumentor>>addMethod: selector:`. If you would need another adaptation add it to the instrumentor and send the message to the instrumentor inside the overridden `adapt:` method
4. The new event defines what kind of information should be reified together with the event to potential observers. Therefore you should specify what kind of arguments you would like the meta-object to provide you with. If your event needs another set of arguments then the parent event needs to override the method `arguments`. In this case you have to add a constructor on the class side with the same arguments and add those as instance variables to the event. Finally you need to override the `reificationBlock` method which returns a block that calls the newly created constructor.

We can also see that by providing a flexible event extension model Chameleon is not bound to any language implementation characteristic. The events granularity can go from a simple variable access to complex conditional expression defined in a new reified event. This approach is equivalent in expressiveness to AOP, Reflex and Reflectivity solutions, as opposed to the Iguana approach with its language limitations.

7.4 Future Work

As future work we plan to validate our approach on more instrumentation-based tools. We plan to build a domain-specific language to ease the definition of events, similar in essence to a pointcut language. We also plan to explore the implementation of the Chameleon approach for other languages. For example, since Reflex was developed in Java this language could be a natural first option. Having meta-events as first class objects allows us to introspect on the events reified during a computation. For example, in the back-in-time debugger scenario we could delay the actual variable analysis until it is necessary and only for specific variables. This might bring performance advantages over fully reflective approaches. We plan to analyze various techniques like this one to take advantage of the event reifications.

Bibliography

[ABB^{+}89] Giuseppe Attardi, Cinzia Bonini, Maria Rosario Boscotrecase, Tito Flagella, and Mauro Gaspari. Metalevel programming in CLOS. In S. Cook, editor, *Proceedings ECOOP '89*, pages 243–256, Nottingham, July 1989. Cambridge University Press.

[AR01] Matthew Arnold and Barbara G. Ryder. A framework for reducing the cost of instrumented code. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, PLDI '01, pages 168–179, New York, NY, USA, 2001. ACM. URL: http://altair.snu.ac.kr/newhome/kr/course/system_software/2005/Arnold01Sampling.pdf, doi:10.1145/378795.378832.

[BGW93] Daniel G. Bobrow, Richard P. Gabriel, and J.L. White. CLOS in context — the shape of the design. In A. Paepcke, editor, *Object-Oriented Programming: the CLOS perspective*, pages 29–61. MIT Press, 1993.

[BNRR11] Alexandre Bergel, Oscar Nierstrasz, Lukas Renggli, and Jorge Ressia. Domain-specific profiling. In *Proceedings of the 49th International Conference on Objects, Models, Components and Patterns (TOOLS'11)*, volume 6705 of *LNCS*, pages 68–82. Springer-Verlag, June 2011. URL: http://scg.unibe.ch/archive/papers/Berg11b-Profiling.pdf, doi:10.1007/978-3-642-21952-8_7.

[Caz98] Walter Cazzola. Evaluation of object-oriented reflective models. In *In Proceedings of ECOOP Workshop on Reflective Object-Oriented Programming and Systems (EWROOPS 98), in 12th European Conference on Object-Oriented Programming (ECOOP 98), Brussels, Belgium, on 20th-24th*, pages 3–540, 1998.

[Coi87] Pierre Cointe. Metaclasses are first class: the ObjVlisp model. In *Proceedings OOPSLA '87, ACM SIGPLAN Notices*, volume 22, pages 156–167, December 1987. URL: http://stephane.ducasse.free.fr/Web/ArchivedLectures/p156-cointe.pdf, doi:10.1145/38765.38822.

[Coi90] Pierre Cointe. The ClassTalk system: A laboratory to study reflection in smalltalk. In *OOPSLA/ECOOP Workshop on Reflection and Metalevel Architecture*, 1990.

[CSL04] Bryan M. Cantrill, Michael W. Shapiro, and Adam H. Leventhal. Dynamic instrumentation of production systems. In *Proceedings of USENIX 2004 Annual Technical Conference*, pages 15–28, Berkeley, CA, USA, 2004. USENIX Association. URL: http://www.usenix.org/event/usenix04/tech/general/cantrill.html.

[DDLM07] Marcus Denker, Stéphane Ducasse, Adrian Lienhard, and Philippe Marschall. Sub-method reflection. In *Journal of Object Technology, Special Issue. Proceedings of TOOLS Europe 2007*, volume 6/9, pages 231–251. ETH, October 2007. URL: http://www.jot.fm/contents/issue_2007_10/paper14.

htmlhttp://www.jot.fm/issues/issue_2007_10/paper14.pdf, doi:10.5381/jot.2007.6.9.a14.

[Den08] Marcus Denker. *Sub-method Structural and Behavioral Reflection*. PhD thesis, University of Bern, May 2008. URL: http://scg.unibe.ch/archive/phd/denker-phd.pdf.

[DMS01] Remi Douence, Olivier Motelet, and Mario Sudholt. A formal definition of crosscuts. In *Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns (Reflection 2001)*, volume 2192 of *Lecture Notes in Computer Science*, pages 170–186, Berlin, Heidelberg, and New York, September 2001. Springer-Verlag. doi:10.1007/3-540-45429-2_13.

[DS02] Rémi Douence and Mario Südholt. A model and a tool for event-based aspect-oriented programming (EAOP). Technical report, Ecole des Mines de Nantes, December 2002.

[Fer89] Jacques Ferber. Computational reflection in class-based object-oriented languages. In *Proceedings OOPSLA '89, ACM SIGPLAN Notices*, volume 24, pages 317–326, October 1989. doi:10.1145/74877.74910.

[GC96] Brendan Gowing and Vinny Cahill. Meta-object protocols for C++: The Iguana approach. Technical report, AAA, 1996.

[GR83] Adele Goldberg and David Robson. *Smalltalk 80: the Language and its Implementation*. Addison Wesley, Reading, Mass., May 1983. URL: http://stephane.ducasse.free.fr/FreeBooks/BlueBook/Bluebook.pdf.

[HJ92] Reed Hastings and Bob Joyce. Purify: Fast detection of memory leaks and access errors, 1992.

[HNM+97] J. K. Hollingsworth, O. Niam, B. P. Miller, Zhichen Xu, M. J. R. Goncalves, and Ling Zheng. MDL: A language and compiler for dynamic program instrumentation. In *Proceedings of the 1997 International Conference on Parallel Architectures and Compilation Techniques*, PACT '97, pages 201–, Washington, DC, USA, 1997. IEEE Computer Society. URL: http://portal.acm.org/citation.cfm?id=522659.825654.

[Ibr91] Mamdouh H. Ibrahim. Reflection and metalevel architectures in object-oriented programming (workshop session). In *OOPSLA/ECOOP '90: Proceedings of the European conference on Object-oriented programming addendum: systems, languages, and applications*, pages 73–80, New York, NY, USA, 1991. ACM Press. doi:10.1145/319016.319050.

[KdRB91] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[KHH+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *Proceedings ECOOP 2001*, number 2072 in LNCS, pages 327–353. Springer Verlag, 2001.

[Kic92] Gregor Kiczales. Towards a new model of abstraction in the engineering of software. In *Proc. of IMSA '92 Workshop on Reflection and Meta-Level Architecture*, 1992.

[Kic96] Gregor Kiczales. Aspect-oriented programming: A position paper from the Xerox PARC aspect-oriented programming project. In Max Muehlhauser, editor, *Special Issues in Object-Oriented Programming*. Dpunkt Verlag, 1996.

[KIL+97] Gregor Kiczales, John Irwin, John Lamping, Jean-Marc Loingtier, Cristina Videira Lopes, Chris Maeda, and Anurag Mendhekar. Aspect-oriented programming. Technical report, Xerox Palo Alto Research Center, 1997.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *ECOOP'97: Proceedings of the 11th European Conference on Object-Oriented Programming*, volume 1241 of *LNCS*, pages 220–242, Jyvaskyla, Finland, June 1997. Springer-Verlag. doi:10.1007/BFb0053381.

[LGN08] Adrian Lienhard, Tudor Gîrba, and Oscar Nierstrasz. Practical object-oriented back-in-time debugging. In *Proceedings of the 22nd European Conference on Object-Oriented Programming (ECOOP'08)*, volume 5142 of *LNCS*, pages 592–615. Springer, 2008. ECOOP distinguished paper award. URL: http://scg.unibe.ch/archive/papers/Lien08bBackInTimeDebugging.pdf, doi:10.1007/978-3-540-70592-5_25.

[McA95] Jeff McAffer. Meta-level programming with CodA. In W. Olthoff, editor, *Proceedings ECOOP '95*, volume 952 of *LNCS*, pages 190–214, Aarhus, Denmark, August 1995. Springer-Verlag. doi:10.1007/3-540-49538-X_10.

[McA96] Jeff McAffer. Engineering the meta level. In Gregor Kiczales, editor, *Proceedings of the 1st International Conference on Metalevel Architectures and Reflection (Reflection 96)*, San Francisco, USA, April 1996.

[MGL06] Michael Meyer, Tudor Gîrba, and Mircea Lungu. Mondrian: An agile visualization framework. In *ACM Symposium on Software Visualization (SoftVis'06)*, pages 135–144, New York, NY, USA, 2006. ACM Press. URL: http://scg.unibe.ch/archive/papers/Meye06aMondrian.pdf, doi:10.1145/1148493.1148513.

[RC00] Barry Redmond and Vinny Cahill. Iguana/J: Towards a dynamic and efficient reflective architecture for Java. In *Proceedings of European Conference on Object-Oriented Programming, workshop on Reflection and Meta-Level Architectures*, 2000.

[RC02] Barry Redmond and Vinny Cahill. Supporting unanticipated dynamic adaptation of application behaviour. In *Proceedings of European Conference on Object-Oriented Programming*, volume 2374, pages 205–230. Springer-Verlag, 2002. doi:10.1007/3-540-47993-7_9.

[Riv96] Fred Rivard. Reflective Facilities in Smalltalk. *Revue Informatik/Informatique, revue des organisations suisses d'informatique. Numéro 1 Février 1996*, February 1996.

[RRGN10] Jorge Ressia, Lukas Renggli, Tudor Gîrba, and Oscar Nierstrasz. Run-time evolution through explicit meta-objects. In *Proceedings of the 5th Workshop on Models@run.time at the ACM/IEEE 13th International Conference on Model Driven Engineering Languages and Systems (MODELS 2010)*, pages 37–48, October 2010. http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-641/. URL: http://scg.unibe.ch/archive/papers/Ress10a-RuntimeEvolution.pdf.

[SE04] Amitabh Srivastava and Alan Eustace. Atom: a system for building customized program analysis tools. *SIGPLAN Not.*, 39:528–539, April 2004. URL: http://doi.acm.org/10.1145/989393.989446, doi:10.1145/989393.989446.

[SK04] Maximilian Störzer and Christian Koppen. Pcdiff: Attacking the fragile pointcut problem, abstract. In *European Interactive Workshop on Aspects in Software*, Berlin, Germany, sep 2004. URL: `http://pp.info.uni-karlsruhe.de/uploads/publikationen/stoerzer04eiwas.pdf`.

[Smi82] Brian Cantwell Smith. *Reflection and Semantics in a Procedural Language*. Ph.D. thesis, MIT, Cambridge, MA, 1982. URL: `http://repository.readscheme.org/ftp/papers/bcsmith-thesis.pdf`.

[Smi84] Brian Cantwell Smith. Reflection and semantics in Lisp. In *Proceedings of POPL '84*, pages 23–3, 1984. `doi:10.1145/800017.800513`.

[SSS00] Omri Traub Stuart, Stuart Schechter, and Michael D. Smith. Ephemeral instrumentation for lightweight program profiling. Technical report, Harvard University, 2000.

[Sun99] Microsystems Sun. *Java Core Reflection API and Specification*, 1999.

[SVJ03] Davy Suvée, Wim Vanderperren, and Viviane Jonckers. Jasco: an aspect-oriented approach tailored for component based software development. In *Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 21–29, New York, NY, USA, 2003. ACM. URL: `http://doi.acm.org/10.1145/643603.643606`, `doi:10.1145/643603.643606`.

[TNCC03] Éric Tanter, Jacques Noyé, Denis Caromel, and Pierre Cointe. Partial behavioral reflection: Spatial and temporal selection of reification. In *Proceedings of OOPSLA '03, ACM SIGPLAN Notices*, pages 27–46, nov 2003. URL: `http://www.dcc.uchile.cl/~etanter/research/publi/2003/tanter-oopsla03.pdf`, `doi:10.1145/949305.949309`.

Appendices

A.1 Installation Guide

Chameleon was implemented in Smalltalk. We choose Pharo as our environment. Pharo is a clean, innovative, open-source Smalltalk-inspired environment. To be able to use Chameleon you will need a Pharo virtual machine, which you can get at `http://www.pharo-project.org/`.

Now you can download the one-click image for your platform from `http://scg.unibe.ch/research/chameleon`. Unzip the file. Afterwards you have to launch the executable of your platform:

```
Mac: chameleon-OneClick.app
Linux: chameleon-OneClick.app/chameleon-OneClick.sh
Windows: chameleon-OneClick.app/chameleon-OneClick.exe
```

Listing 32: Chameleon Executables of each platform.

Another possible way to install Chameleon is to load the code inside Pharo. This might be needed when you want to add Chameleon to an already running Pharo environment. For this purpose you need to execute the following statement:

```
Gofer new
   squeaksource: 'Chameleon';
   package: 'Chameleon';
   load.
```

Listing 33: Loading Chameleon into Pharo.

This will take some time to download and install Chameleon and all its dependencies.